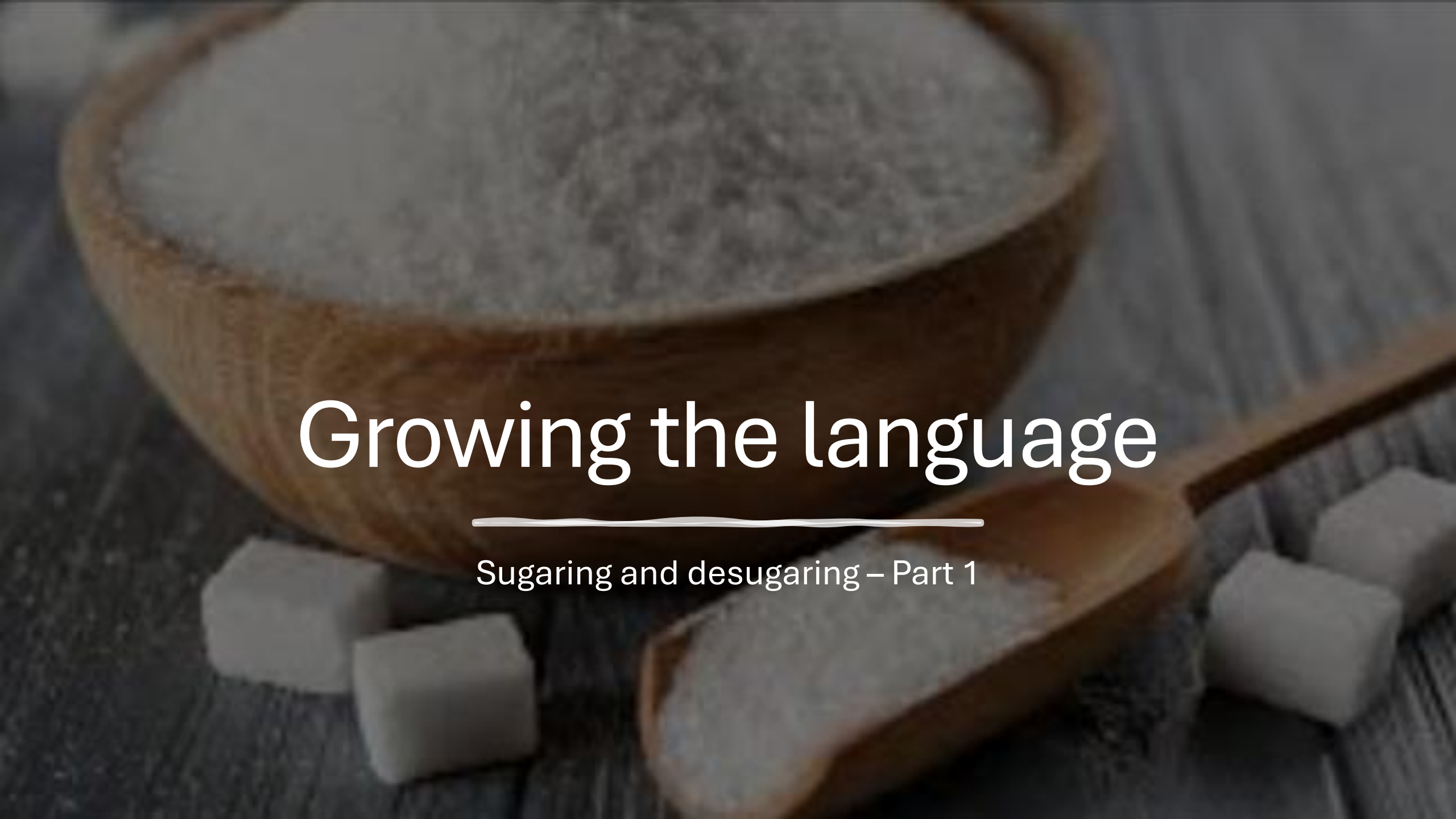


Growing the language

Sugaring and desugaring – Part 1

Growing the language

- How do we give programmers a richer language without making the interpreter more complex?

- We have been following the Standard Implementation Plan (SImPl)

S-Exp → parse → Exp (AST) → interp → Value

What we have so far

Core language	The language What `interp` handles: `numE`, `plusE`, `varE`, `let1E`, `lamE`, `appE`
Surface language	Everything the programmer can write. The core *plus* sugar
Desugarer	A new step between `parse` and `interp` that translates sugar away

Idea (new pipeline)

- A small core language becomes a large surface language through desugaring. Your interpreter stays the same size.

- The surface language can grow indefinitely (including for-loops, whiles, etc).

- The only cost is writing desugar cases. They don't need environments, values, or closures. They just rewrite the tree.

S-Exp → parse → Exp → desugar → Exp (core only AST) → interp → Value

Redundancy in Languages

```
for(x = 0; x < 10; x++) {
  sum += x;
}
```

This is *exactly* the same as

```
x = 0;
while (x < 10) {
  sum += x;
  x++;
}
```

If you implement the while-loop, all of that work has to be duplicated for the while-loop.

Wouldn't it be much simpler to implement it once and translate the for body into a while body?

Desugaring

- Introduces a distinction between a *core* language and a *surface* language.

- The surface language may have various conveniences, but these get translated into the core language, whose constructs are all handled directly

- The extra constructs, those that make it “sweeter” to program, are called *syntactic sugar*

- Translating surface programs down to the core is called *desugarer*

Examples of desugarings in real languages

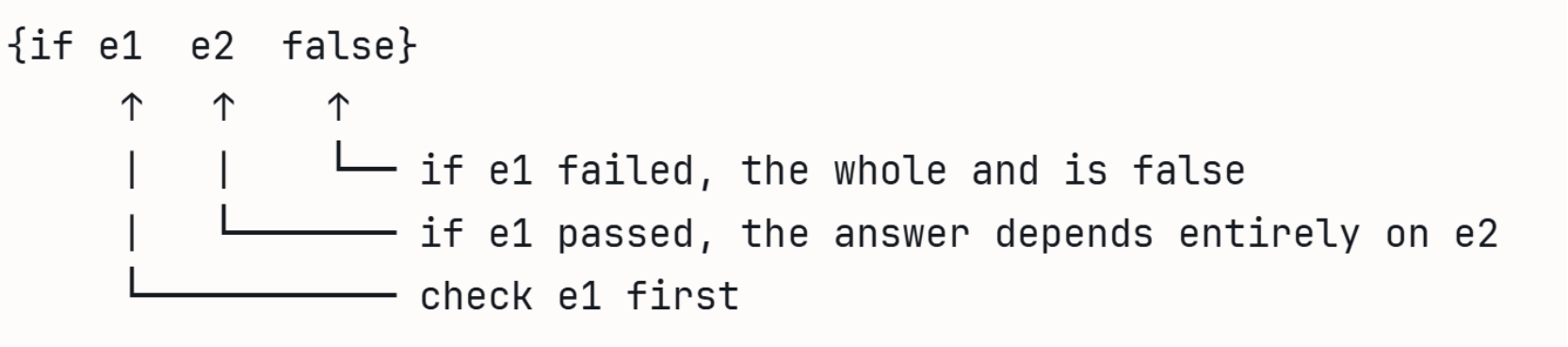
and and *or* can desugar into nested ifs.

{and e1 e2} ->desugar-> {if e1 e2 false}

{ and e1 e2} if e1 is false -> the whole expression is false.

if e1 is true -> the whole expression depends of e2

{if e1 e2 false} When e1 is **true**, the if goes to the **then-branch**, which is e2. So it returns whatever e2 evaluates to:



Examples of desugarings in real languages

{or e1 e2} - >desugar->

```
{let1 {v e1}
  {if v v e2}}
```

{or e1 e2} should:

1. Evaluate e1
2. If e1 is **truthy** → return **e1's value** (not just true)
3. if e1 is **falsy** → return e2

```
{let1 {v e1}        ← evaluate e1 ONCE, store it in v
  {if v             ← test v (not e1 again!)
      v             ← if truthy, return v (the saved value)
      e2}}          ← if falsy, return e2
```

Examples of desugarings in real languages

In many languages, x += y is sugar for x = x + y.

In Python, + desugars into the method __add__.

Python has a whole bunch of these desugarings; these methods are called “dunder” methods

More examples

let1 ->desugar-> lam + app

```
{let1 {x 3} {+ x x}}
; desugars to:
 {{lam x {+ x x}} 3}
```

let1E is purely a convenient syntax on top of them.

How to do it?

- There are two ways to do it.

1. Via functions: Here, we have to rewrite the AST, and they have a lot of problems (see the next example)

1. Via macros: The preferred option.

Using functions

Racket's built-in if is **truthy/falsy,** any non-false value counts as true:

```
(if 0    "yes" "no")   ; => "yes"  -- 0 is truthy in Racket!
(if "hi" "yes" "no")   ; => "yes"  -- strings are truthy
(if #f   "yes" "no")   ; => "no"   -- only #f is false
```

Now, suppose we want a **stricter** version that only accepts actual Booleans (#true or #false) and errors on anything else. How do we build it?

strict-if (as a function)

```
(define (strict-if C T E)
  (if (boolean? C)
      (if C T E)
      (error 'strict-if "expected a boolean")))
```

Tests

(strict-if #true 1 2) ; => 1

(strict-if #false 1 2) ; => 2

(strict-if 0 1 2) ; => error

(strict-if #true 1 (/ 1 0));=> 💥 Your language and Racket are both **eager**: when a function is called, **all arguments are evaluated first**, before the body runs.

(if #true 1 (/ 1 0)) ; => 1, fine — else branch never touched

Built-in **if** is **not a function** — it's a special form that only evaluates the branch it takes.

strict-if (as a macro)

```
(define-syntax strict-if    ; name of the syntax, registers strict-if  as a macro
  (syntax-rules ()          ; defining the rules by pattern matching.  The () is the keyword list. Empty for now
    [(strict-if C T E)      ; pattern to match, C, T, E pattern variables,.
      (if (boolean? C)
          (if C T E)
          (error 'strict-if "expected a boolean"))]))
```

The output: the same code as the function body, but now it's **syntax**, not a function call. It gets spliced into the program source before evaluation

```
(strict-if #true 1 (/ 1 0))
```

The macro expander matches it against the pattern (strict-if C T E)

C = #true
T = 1
E = (/ 1 0); not evaluated yet, just syntax!

Re-writing

```
(define-syntax strict-if          ; 1. name of the new syntax
  (syntax-rules ()                ; 2. no special keywords
    [(strict-if C T E)            ; 3. PATTERN to match
     (if (boolean? C)             ; 4. OUTPUT to produce
         (if C T E)
         (error 'strict-if "expected a boolean"))]))
```

Rewrites the whole thing into:

```
(if (boolean? #true)
    (if #true 1 (/ 1 0))
    (error 'strict-if "expected a boolean"))
```

Functions vs Macro

	Function	Macro
`(strict-if #true 1 2)`	works	works
`(strict-if 0 1 2)`	errors	errors
`(strict-if #true 1 (/ 1 0))`	crashes	works
Why	All args evaluated before body	Syntax rewritten before evaluation

A macro sees your code as **data** and rewrites it into other code. Evaluation only begins after rewriting is complete. That's why it can skip branches that a function never could.

Example 2: my-let1

- We want a local binding form. Specifically, we want:

```
(my-let1 (x 3) (+ x x)) ; => 6
```

To behave exactly like Racket's built-in

```
(let ([x 3]) (+ x x)) ; => 6
```

Try like a function:

```
(define (my-let1 binding body)
  ...)
```

```
Step 1: evaluate (x 3)    ; looks like a function call — x is unbound! 💥
Step 2: evaluate (+ x x)  ; x isn't bound yet anyway! 💥
```

Example 2: my-let1 as macro

- let ->desugar->lambda + immediate application does:

((lambda (x) (+ x x)) 3)

- Applied immediately to 3, so x gets bound to 3
- Body (+ x x) runs in an environment where x = 3

That is identical to let. Both bind x to 3 and evaluate the body. So:

```
(my-let1 (x 3) (+ x x))
; is exactly the same as:
((lambda (x) (+ x x)) 3)
```

This pattern, an anonymous function applied immediately, is called left-left-lambda

Double check in the language

```
; Using let1:
(run '{let1 {x 3} {+ x x}})        ; => (numV 6)

; Using lam + app:
(run '{{lam x {+ x x}} 3})         ; => (numV 6)
```

my-let1 as macro

```
(define-syntax my-let1
  (syntax-rules ()
    [(my-let1 (var val) body)
     ((lambda (var) body) val)]))
```

Dissecting it:

```
(my-let1 (var val) body)     ; PATTERN
                             ;   var  = the name being bound
                             ;   val  = the value expression
                             ;   body = the body expression

((lambda (var) body) val)    ; OUTPUT
                             ;   var  goes into lambda's parameter
                             ;   body goes into lambda's body
                             ;   val  becomes the argument
```

```
(my-let1 (x 3) (+ x x))
```

```
var  = x
val  = 3
body = (+ x x)
```

Output:

((lambda (x) (+ x x)) 3) :

1. (lambda (x) (+ x x)) => a function
2. applied to 3 => x = 3
3. evaluate (+ x x) => (+ 3 3) => 6

Macro 3: my-let2 extension of my-let1

my-let1 binds one name. But Racket's built-in let binds many names at once:

```
(let ([x 3] [y 4]) (+ x y))   ; => 7
```

We want my-let2 to behave the same, namely

```
(my-let2 ([x 3] [y 4]) (+ x y))   ; => 7
```

But, what about 3 bindings, or 4, or 5, or any number of bindings?

Introducing ... the ellipsis (0 or more bindings)

(my-let2 ([x 3] [y 4]) (+ x y))

```
; In the PATTERN:
[var val] ...    ; matches zero or more [var val] pairs

; In the OUTPUT:
var ...          ; expands to all the vars, separated by spaces
val ...          ; expands to all the vals, separated by spaces
```

Think of it like a zip and unzip:

```
([x 3] [y 4] [z 5])
         ↓ zip apart
vars = x y z
vals = 3 4 5
         ↓ use separately in output
(lambda (x y z) body) 3 4 5
```

Macro: my-let2 (multi-binding) using lambdas

```
(define-syntax my-let2
  (syntax-rules ()
    [(my-let2 ([var val] ...) body)
     ((lambda (var ...) body) val ...)]))
```

Dissecting it:

```
([var val] ...)          ; PATTERN: zero or more [var val] pairs
                         ;   var matches each name
                         ;   val matches each value expression

((lambda (var ...) body) val ...)   ; OUTPUT:
                                    ;   var ... = all names as parameters
                                    ;   body    = the body unchanged
                                    ;   val ... = all values as arguments
```

Example

```
(my-let2 ([a 1] [b 2] [c 3]) (+ a (+ b c)))
```

; Macro expander collects

```
Pattern matches:
```

```
var = a, b, c
val = 1, 2, 3
body = (+ a (+ b c))
```

; Macro expander produces this new source code:

```
((lambda (a b c) (+ a (+ b c))) 1 2 3)
```

>>6

The original code **no longer exists**. It has been completely replaced

> `...` lets a macro handle **any number** of sub-patterns — turning a fixed pattern matcher into something that scales to whatever the programmer writes.

Macro 4: *my-cond*

We want a multi-armed conditional, like a chain of if/else if/else

```
(define (sign n)
  (my-cond
    [(< n 0) "negative"]
    [(= n 0) "zero"]
    [(> n 0) "positive"]))
```

Recuse on syntax?

If there are **no arms** -> error, we fell through (base-case)

If there is **at least one arm** -> check its question:

- if truthy -> return its answer
- if falsy -> run *my-cond* on the remining arms. (recursive call)

Macro 4: *my-cond*

```
; Base case — no arms left:
(my-cond)
; => error

; Recursive case — at least one arm:
(my-cond [q0 a0] [q1 a1] ...)
; => (if q0
;        a0
;        (my-cond [q1 a1] ...))
;             ↑
;        recurse on the REST
```

Macro 4: Macro *my-cond*

```
(define-syntax my-cond
  (syntax-rules ()

    ; Base case: no arms left — should never get here
    [(my-cond)
     (error 'my-cond "should not get here")]

    ; Recursive case: peel off first arm, recurse on rest
    [(my-cond [q0 a0] [q1 a1] ...)
     (if q0
         a0
         (my-cond [q1 a1] ...))]))
```

Macro 4: *my-cond*

```
(my-cond
  [(< 3 0) "negative"]
  [(= 3 0) "zero"]
  [(> 3 0) "positive"])
```

Expansion step 1 — recursive case fires, peels off first arm:

```
q0 = (< 3 0)
a0 = "negative"
remaining = [(= 3 0) "zero"] [(> 3 0) "positive"]
```

Produces:

```
(if (< 3 0)
    "negative"
    (my-cond
      [(= 3 0) "zero"]
      [(> 3 0) "positive"]))
```

Expansion step 2 — `my-cond` in the else branch fires again:

```
q0 = (= 3 0)
a0 = "zero"
remaining = [(> 3 0) "positive"]
```

Produces:

```
(if (< 3 0)
    "negative"
    (if (= 3 0)
        "zero"
        (my-cond
          [(> 3 0) "positive"])))
```

Expansion step 3 — `my-cond` fires again:

```
q0 = (> 3 0)
a0 = "positive"
remaining = (nothing)
```

Produces:

```
(if (< 3 0)
    "negative"
    (if (= 3 0)
        "zero"
        (if (> 3 0)
            "positive"
            (my-cond))))
```

```
(if (< 3 0)
    "negative"
    (if (= 3 0)
        "zero"
        (if (> 3 0)
            "positive"
            (error 'my-cond "should not get here"))))
```

No more `my-cond` anywhere. Expansion is complete.

Evaluation — on the fully expanded code:

```
(< 3 0) => #f  → take else branch
(= 3 0) => #f  → take else branch
(> 3 0) => #t  → take then branch → "positive"
```

Result: `"positive"`

The expansion visualized as a tree

```
my-cond
[q0 a0]
[q1 a1]           (if q0 a0
[q2 a2]    =>       (if q1 a1
                      (if q2 a2
                        ERROR)))
```

Macro 4: *my-cond*

```
[(my-cond [q0 a0] [q1 a1] ...)
```

Part	Meaning
`[q0 a0]`	The **first** pair — matched exactly once
`[q1 a1] ...`	**All remaining** pairs — zero or more

```
; my-cond with 3 arms desugars to:
(ifE q0 a0
  (ifE q1 a1
    (ifE q2 a2
      (error ...))))
```

More material

- HTML: 13 Desugaring as a Language Feature
- Video: More on Macros